Experimentation in the Operating System: The Windows Experimentation Platform

Paul Luo Li, Pavel Dmitriev, Huibin Mary Hu, Xiaoyu Chai, Zoran Dimov, Brandon Paddock,
Ying Li, Alex Kirshenbaum, Irina Niculescu, Taj Thoresen
Microsoft
Redmond, USA
{pal, padmitri, maryhu, xicha, zodimov, bpaddock, yinli, alexkir, irnicule, tajt} @microsoft.com

***Abstract*—Online controlled experiments are the gold standard for evaluating improvements and accelerating innovations in online and app worlds. However, little is known about applicability, implementation, and efficacy of experimentation for operating systems (OS), where many features are non-user-facing. In this paper, we present the Windows Experimentation platform (WExp), and insights from implementation and execution of real-world experiments in the OS. We start by discussing the need for experimentation in OS, using real experiments to illustrate the benefits. We then describe the architecture of WExp, focusing on unique considerations in its engineering. Finally, we discuss learnings and challenges from conducting real-world experiments. Our experiences and insights can motivate practitioners to start experimenting as well as to help them to successfully build their experimentation platforms. The learnings can also guide experimenters with best-practices and highlight promising avenues for future research.**

Keywords—experimentation, operating systems, measurement, a/b testing, online controlled experiments

I. Introduction

Online controlled experiments, aka A/B tests, are used by many organizations, e.g. Google, Facebook, Amazon, and Microsoft, to design and improve their apps and online services [1][2][3][4]. However, little is known about experimentation in the operating system, where features being experimented with are often not user facing.

The OS differs from apps and online services in many ways that impact the engineering of an experimentation system as well as the execution of real-world experiments. For example, the interconnectedness of components, the criticality of their functionality, the possibility of offline scenarios, the complexity of data collection, and the process of OS updates all affect experimentation.

For example, consider an experiment designed to improve the OS component that manages the lifetime of Windows processes [5] (e.g. suspending applications, discussed in detail in Section V.B). The feature has no UI, yet it may deeply impact users' experiences. Determining the causal impact of changes (e.g. quality) is critical, but assessments must occur with devices being potentially offline. Furthermore, there needs to be a way to stop the experiment and return devices to the previous behavior faster than shipping code updates to user devices.

Today, there is a dearth of knowledge about how to implement an experimentation platform for the OS, how to execute experiments successfully, and what the limitations/challenges are. To our knowledge, there is no discussion of these topics in the literature. In this paper, we address these gaps in knowledge. Our key contributions are:

- Motivations for starting to experiment in the OS, platforms, and other non-user-facing components
- Practical guidance for implementing a successful experimentation platform for the OS
- Warnings about challenges with experimenting in the OS
- Best practices for executing effective real-world OS experiments

We start by providing background on experimentation in Section II, related work in Section III, and methods in Section IV. In Section V we discuss the need for experiments in the OS, e.g. why the classic approach of extensive testing is insufficient. In Section VI, we present the Windows Experimentation platform (WExp), and discuss key aspects of its design and architecture that are impacted by being for the OS. In Section VII, we discuss lessons learned from executing real-world experiments. We conclude in Section VIII with directions for future work and research.

II. Controlled Experiments

In its simplest form, a controlled experiment (or A/B test) randomly assigns users one of two variants: control (A) or treatment (B). Usually control is the existing system and treatment is the system with a new feature added, say, feature X. User interactions with the two systems are recorded and from that, metrics are computed and compared. If the experiment was designed and executed correctly, the only difference between the two variants is feature X. External factors such as seasonality, impact of other feature changes, moves by competition, etc. are distributed evenly between control and treatment, and therefore do not impact results of the experiment. Hence, differences in metrics between the two groups can be attributed to either feature X or noise. The noise option is ruled out using statistical tests (e.g. the t-test [6]). The upshot is that a causal relationship between the change to the product and changes in user behavior is established through the experiment. Experimentation has been shown to be especially useful when user reactions are uncertain for design changes or novel innovations [2], providing organizations an effective approach of making data-driven decisions to improve their products and services. In this paper, by experiments we mean *randomized controlled experiments*.

III. Related Work

Related work falls into two broad categories: publications describing the architecture and the engineering/quality assurance processes of commercial operating systems, and papers on randomized controlled experiments.

Many researchers and practitioners have reported on various aspects of the engineering and validation of large-scale widely-used commercial operating systems. Studies have examined various areas in detail, such as the kernel [7], the file system [8], the error reporting system [9], cryptography [10], IP protocol [11], the driver model [12], and the memory system [13]. There are also numerous publications about engineering and quality assurance processes, including distributed development [14], detecting race conditions [15], debugging [16], and reliability measurement [17]. However, no prior work has discussed experimentation in real-world operating systems.

Experimentation is an active research area, fueled by the growing importance of online experiments in the software industry. Research has been focused on topics such as new statistical methods to improve metric sensitivity [18] [19], metric design and interpretation [20], [21], projections of results from a short-term experiment to the long term [22], [23], the benefits of experimentation at scale [24], [25], experiments in social networks [26], high-level architecture of an experimentation platform [27], as well as examples, pitfalls, rules of thumb and lessons learned from running controlled experiments in practical settings [21], [28], [29]. These works provide good context for our paper and highlight the importance of having a solid experimentation platform to benefit a variety of software engineering practices.

In this paper, we discuss the unique challenges of experimentation in OS, one of the most complex scenarios to apply experimentation to. We also share the insights and lessons from executing real-world Windows OS experiments.

IV. Method

Our case study of the Windows Experimentation Platform (WExp) covers work at Microsoft Corporation in the USA between 2016 and 2018. The authors of this paper are or have been employed at Microsoft and are the primary engineers and data scientists who implemented the WExp system and oversaw experiments. We briefly describe our approach as prescribed by Runeson and Höst [30].

A. Data Collection

We leveraged three primary sources of data. First, we used data from real Windows experiments, including analysis reports by engineering teams and data scientists, which the authors had access to as co-authors. These contained findings, conclusions, and subsequent engineering decisions. Though numerical results are omitted, we provide actual feature names as well as insights/conclusions from those experiments. Second, we used source code and documentation (e.g. notes, emails, presentations, and design docs) both for the WExp system and features using WExp, extracted from source control systems and document depots (e.g. Sharepoint). We share actual Windows code (redacting some product specific information) and the design intent of the features. Finally, we also used emails and meeting notes from triaging and diagnoses of issues that occurred during the experiments. From these we provide insights on best practices as well as challenging problems.

B. Data Analysis

Overall, we present factual findings and subsequent engineering outcomes. We report actual implementation decisions and their implications. We discuss issues encountered in the execution of experiments. Our analyses and insights are based on the authors' knowledge/experience with Windows (e.g. [31]), experimentation (e.g. [25]), and the literature.

C. Threats to Validity.

To reduce the risk to construct validity, we name features, components, and findings (where possible). We use industry standard terminology and provide explanations/examples to clarify ambiguous concepts and terms. Though we omit numerical results, to reduce threats to internal validity, this paper has been reviewed internally to ensure accuracy. Finally, though this paper is a case-study with external validity limitations, many of the topics and concerns are not tied directly to Windows and are applicable to other contexts and organizations.

V. Motivation: Why OS Experimentation?

Related work shows that experimentation enables organizations to innovate effectively; however, experimentation discussed in the literature is typically for user-facing scenarios, where there is uncertainty in user reactions (thus the need to experiment, measure, and compare). In contrast, behaviors and features of an OS are often not user-facing and are supposed to be deterministic and predictable. Therefore, a foundational question is: what are the benefits of experimenting in the OS? Where are the sources of uncertainty for non-user-facing features? And especially for new features, why is the traditional approach of extensive feature/system testing insufficient? In the following sections, we discuss five inter-related reasons for experimenting in the OS, using actual Windows experiments to explain the benefits. The reasonings and insights are summarized in Table 1.

A. Myriad Software, Hardware, And Scenario Combinations

As discussed in numerous publications about Windows [9] [17], the surface area of an OS can be enormous. Various

Table 1. Summary of benefits of experimentation in the OS

Need	***Explanation***	***Benefits of experimentation***
Myriad software, hardware, and scenario combinations	Complete testing of all software, hardware, and scenarios combinations in-house is impractical	Assess impact of changes in more contexts during development
Isolating impact of a change amidst many concurrent changes	Many concurrent changes create difficulties in attributing issues to a specific change	Isolate the impact of a single change to enable necessary adjustments during development
Assessing trade-offs	Many features entail trade-offs between multiple desirable attributes	Assess trade-offs and make informed data-driven decisions about features
Limiting the impact of failures	Impact of failures can be very high in the OS	Exposure control and recall; limiting potential downsides until impacts of changes are understood
User-facing OS features	Many parts of the OS *are* user-facing, e.g. navigation menus	As with app and online scenarios, assess uncertain user reactions to changes

hardware and software configurations as well as the scenarios in which they are used (e.g. combinations of apps) can cause unforeseen behaviors, e.g. crashes and hangs [17]. Complete testing of all combinations in-house is impractical; experimentation can help to assess impact of changes in more contexts during development.

An example of such a feature was the Host Activity Manager (HAM). HAM aimed to replace the Process Lifetime Manager (PLM) and several other services for managing the lifecycles for Universal Windows Platform (UWP) applications (e.g. suspend and resume). HAM consolidated functionalities, simplified the architecture and API, as well as established a more robust code foundation for further development of UWP.

Since UWP aimed to abstract away differences in the underlying hardware [5], HAM had to handle diverse hardware configurations. Furthermore, since HAM is a foundational component for all UWP apps (e.g. apps in the Windows Store) myriad usage scenarios would have to be validated. Therefore, during development, in addition to extensive internal testing, the team also conducted an experiment on Windows Insiders [32], the pre-release user population, to evaluate the feature in more software, hardware, and app combinations world-wide.

The experiment discovered new issues, as well as exposed the increased severity of some known issues in real-world scenarios. Furthermore, the experiment allowed the team to quickly isolate problems, mostly through bypassing anomalies due to bad devices and badly coded apps, as those issues were mostly equally present in both control and treatment conditions (discussed more in the next section). The team fixed issues and used further experiments to validate/measure improvements, ultimately shipping a high-quality HAM.

Without experimentation, the team would have had difficulty ensuring quality with myriad software, hardware, and app combinations. Also, internal testing alone would have exposed users to more risk, as there would be no way to quickly recall the feature (without shipping new code) should new combinations reveal critical quality issues (discussed more in Section V.D).

B. *Isolating Impact Of A Change Amidst Many Concurrent Changes*

Many features, improvements, and fixes go into a release of an OS. The concurrent nature of these changes creates difficulties in attributing issues to a specific change. This can be particularly problematic when an innovation has the potential to destabilize the system. Experimentation can help engineers isolate the impact of a single change, amidst concurrent changes, and make necessary adjustments during development.

One example of such a feature was Background Activity Moderator (BAM). Minimizing power consumption while maximizing responsiveness is a significant engineering challenge. The BAM feature aimed to achieve this duality, by first categorizing processes as important or unimportant, based on the nature of processes (e.g. background activities). Then, to reduce power consumption while maintaining responsiveness of user interactions, BAM throttles unimportant processes in one of two ways:

- Processor Power Management: limiting the maximum processor frequency of the process.
- Timer Stretch: delaying the processor timer. Individual apps needed to call the API to be opted in. Chrome, Outlook, Word, PowerPoint, Excel, and OneNote were part of the experiment.

The primary focus of the BAM feature was to improve battery life without impacting performance or reliability. However, concurrently with BAM development, several other Windows features were also aiming to improve battery life, and many other features in development impacted quality. Furthermore, there were also ongoing changes in the applications themselves (e.g. the Office apps). To isolate and evaluate the effectiveness of BAM, as well as to identify possible quality issues, the team ran multiple experiments on Windows Insiders during the development of this feature.

Using experiments, which isolated the impact of BAM (using randomization and a control group), the team was able to quantify improvements to power consumption due to BAM, as well as validate no regressions in performance or reliability.

Isolating the impact (positive and negative) of the feature is perhaps the most important benefit of experimentation, Otherwise, addressing issues and shipping a high-quality feature (concurrently with other features) could be a much more time-consuming (and potentially wasteful) process, due effort needed to investigate (potentially irrelevant) issues caused by other features, slowing the pace of innovation.

C. *Assessing Trade-offs*

While many features and behaviors of the OS may be deterministic, their impact may not be clear. Many entail tradeoffs between multiple desirable attributes (e.g. performance and battery life), which given real-world constraints may not all be achievable simultaneously. Experimentation allows teams to assess the trade-offs and to make informed data-driven decisions about their feature.

One example of such a feature was Extra Delay Background Transport (LEDBAT). LEDBAT is a delay-based congestion

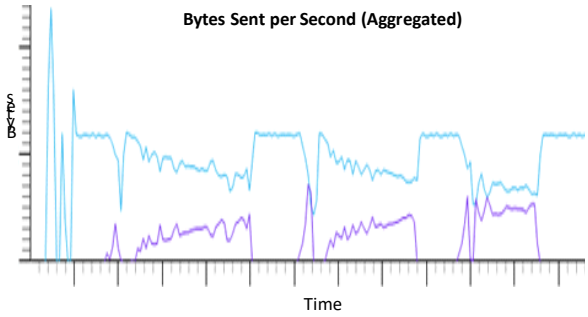

Figure 1. Network traffic without LEDBAT throttling

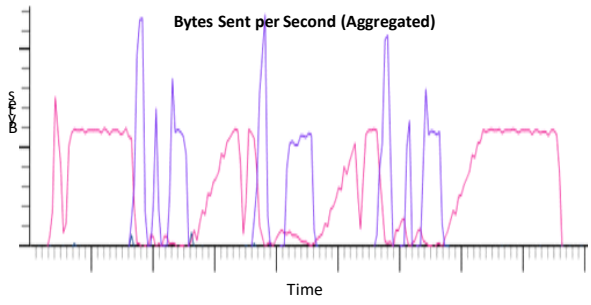

Figure 2. Network traffic with LEDBAT throttling

control algorithm. It improves users' networking experience by throttling background activities that require internet connection, such that the bandwidth allocated to foreground activities are greater. Figure 1 and Figure 2 show the networking bandwidth of the foreground process, in purple, without and with LEDBAT in lab tests. Without LEDBAT, the background process, in blue, contends with the foreground process reducing the bandwidth of the foreground process. With LEDBAT, the background process, in red, is throttled whenever there is networking activity in the foreground process, improving the experience for users.

A potential drawback of this feature is that the throttled background process may have trouble completing its task. In the Windows Insiders experiment, the throttled activity was Windows Error Reporting (WER) [9]. WER uploads debugging information for crashes/hangs, which may use substantial bandwidth. However, Windows engineering teams depend on this debugging information to fix issues and improve user experiences, especially during development. Throttling WER may cause uploads to not complete and impact this critical quality improvement process. The team used experiments to understand the trade-offs between improvements in network throughput and the potential loss in WER upload completes.

Fortunately for this experiment, LEDBAT improved the throughput of non-WER processes without statistically significantly degrading WER upload complete rates. WER's retry logic enabled most uploads to complete, eventually.

Nonetheless, reaching this conclusion about the trade-offs and making an informed data-driven decision about LEDBAT/WER would not have been possible without experimentation.

D. Limiting The Impact Of Failures

Unlike online services, the impact of failures can be very high in the OS. Online services can revert to known good behavior quickly and effectively [33]; problems in the OS can be highly impactful and difficult to remediate (e.g. shipping updates [34]). Experimentation can behave as exposure control and recall tool, limiting the potential downsides of innovations until the impact of changes are thoroughly understood.

One example of this scenario was the TCP Fast Open feature. TCP Fast Open (TFO) is a feature that reduces the time needed

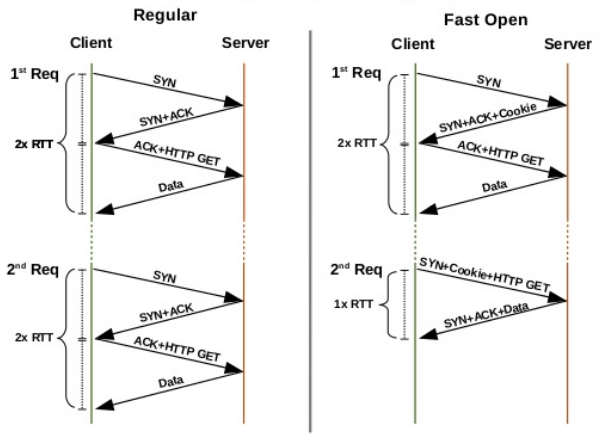

Figure 3. TCP Fast Open savings in network round trips

to open successive TCP connections between two endpoints [35]. TFO uses a cryptographic cookie stored on the client side to reduce a round trips during reconnects (see Figure 3).

The team initially shipped TFO enabled by default but received user feedback about browsing issues, caused by lack of infrastructure support in various parts of the world. Since users spend a majority of their time browsing, the team eventually rolled back the feature. The team then proceeded to progressively experiment with specific websites and in various regions of the world, seeking to limit the possible downside of TFO, while progressively getting the feature ready to be shipped again.

The TCP Fast Open feature demonstrated both the benefits of using experiments to manage the release of features, as well as the potential downside risk of releasing features without experimentation.

E. User-Facing OS Features

Finally, while parts of the OS (e.g. networking and app platform) are hidden from the user, many other parts of the OS are user-facing. From navigation menus, to system alerts, to settings, many OS features are user-facing, and changes in these features can drastically impact user experiences. Consequently, some OS features have as much need for experimentation as user-facing online services or apps.

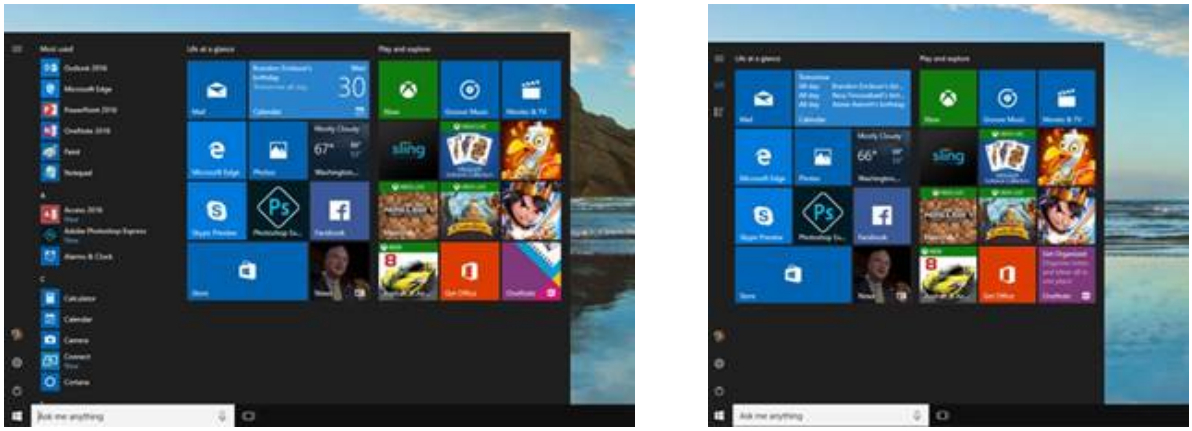

Figure 4. Start Menu experiment setup

One example of such a feature is the default app listing setting in the Start Menu. The team received user feedback asking for the option to hide the app list on the Start Menu. The engineering team built this feature but debated whether to have it set as default (See Figure 4).

As with online and app experiments discussed in the literature [2], the decision was not obvious. While the feature was in response to user demands and may increase usage of UWP apps, removing the app list represents a large departure from the Windows 7 Start Menu interface. The team used a Windows Insider experiment to assess the design choice.

Leveraging both quantitative and qualitative metrics, the team found that the control was better (discussed more in Section VII.A). Insiders were having difficulty navigating the Start Menu without the app list. As a result, while the team shipped the option to hide the app list, the default was set to keep the list visible. Without experimentation, the team's intuition may have resulted in a suboptimal user experience.

VI. Implementation: The Windows Experimentation Platform (WExp)

The previous section explained how experimentation helped Windows teams make confident data-driven decisions about their features/innovations, which we hope motivates practitioners to start to experimenting in their OS, platforms, and other non-user-facing components. In constructing their own experimentation systems, practitioners can benefit from our experiences overcoming unique challenges in implementing an experimentation platform for Windows.

Commercially available solutions and experimentation platforms in the literature are largely server-side systems (e.g. web-pages) where either control or treatment behavior is delivered to the user at the time when the experimental scenario is encountered, and complete data are captured server side. In an operating system, due to the need to quickly terminate experiments and return devices to known safe behaviors, the code/behavior for both control and treatments are delivered to devices, then a centralized service manages experimental assignments. Due to possible offline scenarios and telemetry delays, the inbox parts of the system record and cache information about the assignments, exposures (i.e. actual executions of the code paths), and data about user/system behaviors. Finally, since information from other interacting services (e.g. Bing) may be important for evaluating the success of experiments, the system performs fuzzy joins from various data sources to produce the final analysis.

The most important design consideration is safety. Recovery for server-side systems is generally straight-forward: the server-side system reverts to previously known good configurations (e.g. switch traffic to a backup [33].). Subsequent user requests (including reloads) are back to normal. Most problems in the interim manifest similarly to common connectivity problems (e.g. 404 errors), which most applications and users are resilient to. In contrast, for an OS, deploying new code to client devices is difficult and failures can have undesirable consequences. To ensure safety, several parts of WExp work in concert to ensure that Windows experiments are safe by construction. Coding of experiments (Section VI.A) ensures that both control (the safe fallback) and treatment code are deployed (i.e. no addition code deployment needed for recall). Centralized management of experiments (Section VI.C) enables remote recall and prevents subsequent execution of treatment code. Finally, built-in expiry dates (Section VI.D) ensures returning to the safe fallback even in offline scenarios.

WExp leverages several existing Microsoft components/services, uses others in novel ways, and has new components/services. Delivery of code to devices leverages the Windows Update service [36] using standard processes. Data is recorded and transmitted using Windows Telemetry services and adheres to user settings and restrictions [37]. Randomization and statistical comparison of data are performed using the ExP system [27]. Other than these existing components and services, WExp has modified or created several components, which we discuss in the following sections. We discuss the interesting aspects of WExp using the Start Menu experiment as example, following it through its experimentation lifecycle:

1. Coding experimental variants
2. Creating measures
3. Launching the experiment
4. Executing and monitoring
5. Analyzing results

A. Coding Experimental Variants

OS experimentation starts with the creation of one or more "variants" code paths, where each variant is an adjustment to the behavior of the system. To declare the existence of variants to the experimentation platform, variants are first defined using a 'feature XML', to be included with the code and ingested by the WExp system. Relevant parts of XML declaration for the Start Menu experiment is in Figure 5.

```
01: <feature>
02:  <className>
     Feature_StartSplitViewInMenuMode
     </className>
03:  <id>[FeatureId]</id>
04:  <name>
     Enable view selection in menu mode
     </name>
05:  <description>
       In desktop menu mode, Start now shows
       Either tiles or all apps instead of
       showing both at the same time
     </description>
06:  ...
07:  <variants>
08:    <variant>
09:      <id>1</id>
10:      <enumName>
         HideAppListOnByDefault
         </enumName>
11:      <name>
         Hide app list setting on by default
         </name>
12:      <description>
           Hide app list system setting for
           Showing the nav pane in menu mode
           is on by default
         </description>
13:    </variant>
14:  </variants>
15: </feature>
```

Figure 5. XML Experiment definition

```
01: bool StartData::IsHideAppListEnabled::get()
02: {
03:   bool hideAppList = false;
04:   Feature_StartSplitViewInMenuMode::
      RunIfEnabled(WI_DIAGNOSTICS_INFO, [&]()
05:     {
06:       HRESULT hr = m_startModel->
          GetDefaultUserSetting(&hideAppList);
07:       if (hr == E_NOT_SET)
08:       {
09:          // The user hasn't explicitly
             chosen a value for this setting.
10:          // Use the variant to determine
             the default value to use.
11:          hr = S_OK;
12:          hideAppList =
             Feature_StartSplitViewInMenuMode::
             IsVariantEqual(
             Variant_StartSplitViewInMenuMode::
             HideAppListOnByDefault);
14:       }
15:      return hr;

16:   });
17   return hideAppList;
18: }
```

Figure 6. Check for experimental condition

The key aspects of the XML are the unique [FeatureId], line 3, and the declared variant ("Hide app list setting on by default") line 08-14. Every feature also implicitly has a default "variant 0" path. This default—the currently shipping behavior of the feature—is the code path executed by both non-participants in the experiment and the control group. Having this default behavior—the safe fallback—alongside variant behaviors is an important aspect of WExp.

In the code itself, engineers check for variants. Figure 6 shows the relevant parts of the code implementation for the Start Menu experiment. The variant is checked on line 12, using the name of the feature and the name of the variant. A key implementation detail is checking for filtering conditions prior to checking for the variant on line 06-07. Checking all filtering conditions prior to experimental assignment is a requirement common to all experiment system [27]. However, in online experiments, applicability checks are typically done server side, and then either only the control or the treatment condition is sent back to the user/client. Thus, the client-side code (e.g. web page) is oblivious to applicability conditions and simply renders the content. For WExp, while targeting (discussed more in Section VI.C) tries to be as precise as possible about the devices that receive experiments, there is still a possibility that between receipt of the experiment and execution of the code path, conditions change—user sets/changes the Start Menu default list view setting—to exclude the device from the experiment. Since this gap could be hours or days, for OS experiments, any filtering condition that can be changed needs to be verified in the code prior to applying the experimental assignment.

```
https://[webservicelocation].com?
OEMModel=HP%20Z420%20Workstation&
FlightRing=Canary&
InstallLanguage=en-US&
IsDomainJoined=1&
OSSkuId=4&
OSArchitecture=AMD64&
OSVersion=[OSBuild]&
DeviceFamily=Windows.Desktop
```

Figure 7. Call up to experimentation system

B. Measures: Quantitative And Qualitative

Windows follows best practices in creating metrics for experiments [38]. Teams establish success and quality metrics that holistically assess their features/changes. In addition, the WExp system also provides a set of guardrail metrics to ensure that no experiment negatively impacts key aspects of the system, e.g. quality and user experience.

A distinguishing feature of the WExp system is integration of qualitative feedback. By leveraging the Windows feedback mechanism [39], experimenters can directly query users in the experiment about their experiences (asked in a general way about the scenario, applicable to both control and treatment conditions, not biasing results), in addition to collecting quantitative metrics about user behaviors. This information provides contextual understanding about user behaviors: the *why*, in addition to the *what*. This has proven valuable for many experiments; Section VII.A discusses benefits of qualitative feedback for the Start Menu experiment.

C. Centralized Management

Experiments are launched centrally from a web portal. When devices call up into the experimentation service, information about devices are examined for applicability of experiments. These include static attribute checks (e.g. build and device family), as well as inclusion in computed groups. An excerpt of the call up to the service, with the static attributes, is shown in Figure 7. Applicable devices are then randomized into control and treatment conditions, and the appropriate configuration is sent back to the device. A sequence diagram of the lifetime of the call to the service is shown in Figure 8.

A key architectural decision for the WExp system was whether experiments would be controlled server-side or client-side (i.e. the code itself randomizes into control/treatments, and then send the assignment information back). Three considerations factored into choosing server-side management. The first factor is computed groups. Many experiments are intended for special groups, such as devices that have used specific features/apps in the past. While it is theoretically possible to cache information and compute applicability on the client, potentially large storage needs for past data and complexity of computations makes client-side solutions undesirable.

The second factor is the need to quickly recall experiments. In a client-side solution, complete recall of an experiment would require shipping and installing code updates, which is difficult to do quickly for an OS (even during development). Server-side control keeps devices in the default—safe fallback—condition, until assigned to the variant, controlled by the experimentation service. A centrally issued recall can then switch devices back to the safe fallback condition, while not letting new devices run the variant code (i.e. stop sending assignments).

Finally, having a centralized service (i.e. server-side control) ensures consistency and simplifies improvements. Fixes and improvement, e.g. updates to the randomization algorithm, can be made in one place. Furthermore, reporting of execution of the

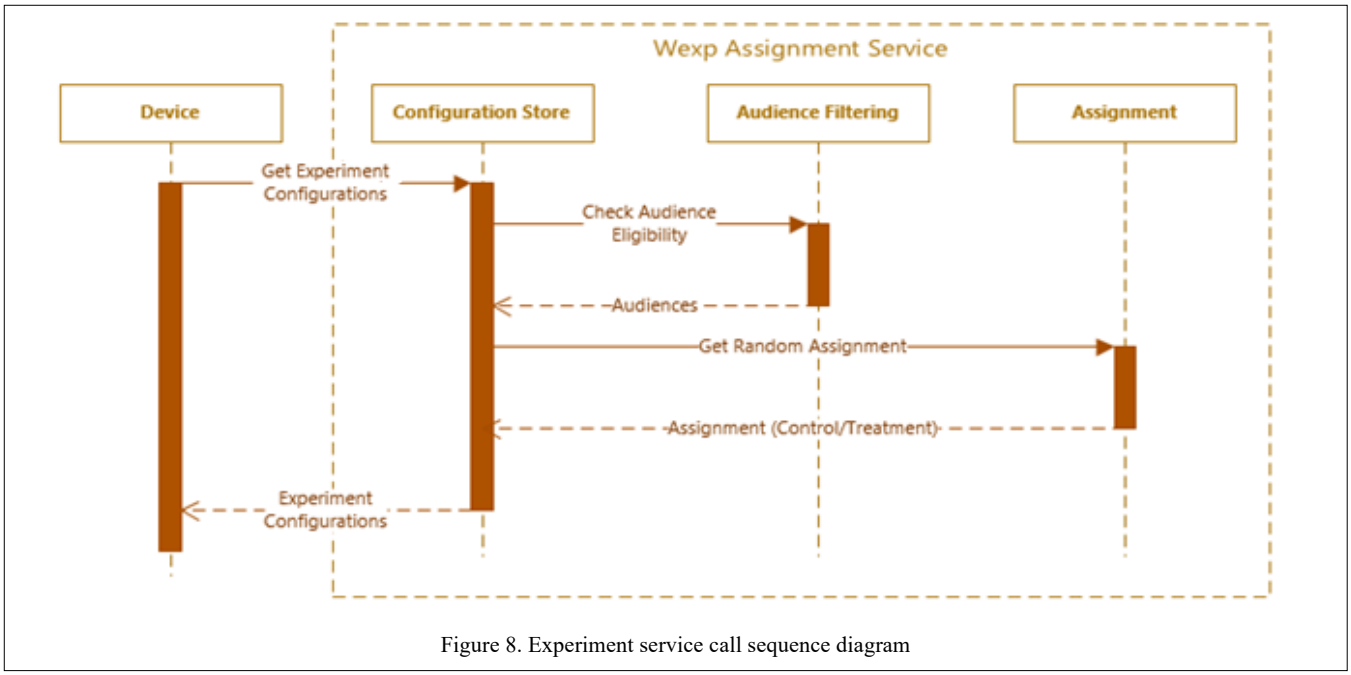

Figure 8. Experiment service call sequence diagram

experimental code paths is consistent, as we discuss in the next section.

D. *Stored Assignment Information And Unified Exposure Telemetry*

Once a device receives its experiment assignments, the information is cached in the OS, thus reducing performance concerns and enabling offline experimentation. To all experimentation code in the OS, the call for assignment information (e.g. line 12 in Figure 6) returns instantaneously and consistently regardless of connectivity (i.e. the inbox part of WExp mimics a constantly available assignment service). The call is a read from the WExp inbox service, making it usable in offline scenarios and even in process/app start-up scenarios. Figure 9 contains an excerpt from the assignment payload with the [FeatureId] and variant assignment on line 07.

WExp gets regular updates from the assignment service. Furthermore, assignments also have expiration dates (line 12 of Figure 9), such that eventually all experiments will revert to the safe fallback.

Another aspect of the WExp system is automatic exposure recording. The exposure information is recorded by the WExp system, based on when the experiment assignment information is read and used in the code. In this manner, all experiments in the OS have a centralized place where exposure is recorded. The information is also stored on the system if the device is temporarily offline or otherwise cannot send telemetry data back. This helps to ensure consistency in reporting across different parts of the OS and to have complete information about all OS experiments on the device (see Section VII.E for more discussion).

```
01: {
02:   "v": "1.0",
03:   "ad": {
04:     "entityId": "[ExperimentIdentifier]",
05:     "items": [
06:       {
07:         "featureId": [FeatureId],
            "variant": 1
08:       }
09:     ]
10:   },
11:   "prm": {
12: "startTime":"[Start DateTime]",
    "expireTime":"[Expire DateTime]"
13:  }
```

Figure 9. Excerpt from assignment payload

E. *Metrics Service*

Every feature in the OS can have its own success/quality metrics, entailing various telemetry data. One challenge is ensuring that all necessary telemetry is sent back for analysis, while not sending more data than needed. WExp leverages existing controls in Windows data collection processes to ensures that only the relevant data are collected, consistent with existing user settings.

Once telemetry data about exposures and other user/system behaviors are sent, WExp performs various data processing and reconciliation tasks to enable comparison of metrics using standard statistical techniques. Processing of metrics and selecting the appropriate datetime ranges to include in the computations (since the date/time when each device executes the experimental code paths can differ) is fairly standard [27].

Data reconciliation across the device and other services, however, is more nuanced. Since Windows often interacts with other apps and services (e.g. Bing), there are identifier and timing challenges. Different apps/services due to their scenarios, may use different identifiers to track their data. Also, because the OS has telemetry delays (e.g. when devices are offline), the reconciliation process needs to account for incomplete data. To deal with these issues, at the final analysis time, WExp first gathers the latest identifier look-up information from other services and then processes the data. That is, since telemetry data from devices can have delays, to have the most complete data for analysis, there needs to be a data reconciliation process.

VII. Lessons And Challenges

In this section, we discuss key lessons learned and challenges encountered when executing experiments in the OS. These can help practitioners execute better experiments and help researchers focus their research on real-world problems.

A. Qualitative Complements Quantitative

For OS experiments, teams use various measures to assess the success and quality of their feature. We have found qualitative feedback to be important in complementing quantitative measures.

All experiments have guardrail metrics, discussed in Section VI.B; each experiment also has feature success and quality metrics. For example, in the Start Menu experiment, the success/quality metrics included Start Menu usage time, Start Menu dwell time, proportion with Start Menu personalization, number of Start Menu invocations, UWP usage time, number of UWP apps used, as well as crashes/hangs.

WExp also empowers teams to solicit user feedback, a qualitative measure, specific to their scenarios. Based on system triggers (e.g. completion of a Start Menu invocation after exposure to the control/treatment condition) the system prompts users with a question about their experiences. For the Start Menu, the question was a 5-point rating for their experience with the Start Menu, as well as optional free text to explain their ratings. The question is phrased generically about the overall scenario, without referencing the feature (e.g. the task list) to avoid biasing.

For the Start Menu experiment, the responses were instrumental to the final decision. The team observed that many quantitative feature success metrics (e.g. Start Menu usage time) increased, which would have indicated success; but, the qualitative feedback told a different story. Ratings were worse in the treatment. Verbatim indicated that users were spending more time in the Start Menu because they were having trouble locating their apps and were trying to get the app list back. Based on this qualitative data, the team decided to keep the behavior of having the app list visible by default.

In addition to enriching the *what* of quantitative metrics with the *why*, user feedback is also a catchall for issues impacting the experiment. Feedback has identified problems with the experiment setup (e.g. the code not behaving as expected under various conditions) and external factors impacting outcomes (e.g. large issues in other parts of the system that drown out the effects of the experiment). Teams have used the information to fix their experiments or design new experiments that address more pressing issues. Overall, qualitative feedback has proven valuable in running successful OS experiments.

Qualitative feedback is rarely discussed in the experimentation literature, possibly due to the lack of built-in support for gathering such data (e.g. the availability of an integrated feedback system). Nonetheless, based on our experiences, we recommend that practitioners collect these valuable data for their experiments.

B. Experimenting On Pre-Release Populations

Windows Insiders differ from General users in many ways, and results/findings from Insiders experiments may not extrapolate [17]. Nonetheless, we have found experimentation on Windows Insiders to be valuable for four reasons, as we explain in the subsequent sections.

1) Positive reactions (or the lack thereof) may be dubious, but negative reactions are likely actionable

Windows Insiders are more tech savvy then General users. Therefore, while claiming success of features based on Insider experiments is questionable, if Insiders have trouble with a feature, then the feature is probably problematic.

The Start Menu default app listing experiment, discussed in Section VI.E and Section VII.A is such an example. Quantitative and qualitative data combined to indicate that even tech savvy Insiders were having trouble locating their apps and were unable to get the list back. Consequently, the team kept the default to have the app list visible.

While Insider experiments may not be conclusive about positive reactions, they have proven effective in discovering *poor* choices, enabling teams to make decisions.

2) Results from system-centric experiments are likely credible, even on Insiders

Some OS experiments aim to assess *system* behaviors. As discussed in Section V.A uncertainty in these experiments are due to diverse software, hardware, and scenario combinations. For features that are system centric (i.e. were user reactions are not as important) Insider experiments may be sufficient.

The Seven-Step Brightness experiment is an example. The Action Center team ran an experiment to assess the benefits of moving to a seven-step screen brightness selector (instead of the default 5). Not having enough hardware diversity or sample size internally, the team ran an experiment on Insiders. Results showed that screen brightness settings were different between Microsoft devices and other devices. A split of the data between those configurations found *negative* impact for Microsoft devices in high power-drain scenarios. The team decided to pull back the feature, to avoid degrading battery-life.

For system-centric changes, experiments on Insiders have proven to be generally sufficient for making decisions.

3) Ensuring quality

As discussed in Section V, problems in OS features can be high impact, and with myriad concurrent changes, accurate accessing impact can be difficult. Experimentation is an effective quality assurance tool that team can utilize.

Edge Pre-Launch experiments are good examples of the value of this approach. The Edge team experimented with launching Edge in the background upon start-up to improve responsiveness of a subsequent user launch. The first iterations of the experiment on Insiders revealed increases in failures due to launching Edge close to start up. Many of the issues were not new (since the scenarios can occur naturally) and may be obscured amidst other changes in Edge. The experiment allowed the team to isolate and address key issues, enabling them to ship a high-quality feature.

4) Making an (initial) data-driven decision

A data-driven decision from an experiment, even on Windows Insiders, is better than guessing. When a change must be made to the software product, using an Insider experiment to

arrive at an initial decision can be an effective approach to accelerate innovations.

The Edge Hub Icon experiment is one such example. Usage data and user feedback indicated that users were not engaging with the Edge Hub (the dropdown menu on the title bar, shown in Figure 10) or were looking for its functionalities in other places. Indications were that the Hub icon was not intuitive, causing users confusion. The team created several design alternatives; rather than guessing, the team used an Insider experiment to select the best design [40].

Data indicated that there were several clear losers and a couple of likely winners. The team then selected the best one to ship. Data from the experiment on pre-release users enabled the team to use a data-driven approach in making the decision.

C. Data Delay And Data Loss

Various reasons can lead to data delays or data not being sent. Such delays and losses can cause unpredictable biases and difficulties for experimental analysis [41].

The Windows OS (and many apps) can operate offline. By design, OS experiments will persist even in these scenarios. However, the data from user/system behaviors cannot be sent immediately back to Microsoft. Furthermore, even if connected, due to consideration to user experiences, data are not always immediately uploaded.

Data delays and data loss (in the cases when devices never come back online) can lead to several biases. The first common type of bias occurs when data loss is directly correlated with treatment itself, i.e. treatment impacts loss rates. The unbalanced loss rates between treatment and control variants result in a biased comparison and can render experiment results invalid. One approach we have taken to mitigate this issue is to add additional metrics around the data loss (i.e. rate of devices not reporting data in control and treatments). This helps to identify the presence of this issue and to include the delay/loss information into the decision making for the feature. Furthermore, when appropriate, we also impute the value of the metrics (e.g. nulls as zeros) to facilitate valid comparisons.

The second type of bias occurs when the population of users that have more data delay/loss react differently to treatments, i.e. there is an interaction. For example, data delay/loss can occur more frequently for casual users (e.g. users that do not use their devices every day). Reaction of those users and behaviors of features under those conditions may differ from other users. Consequently, analysis that do not include the complete data, may have unknown biases. This issue is especially challenging because the typical approach of waiting for more complete data (which may take weeks) and repeating the analysis may not be timely for engineering teams seeking to make decisions about their features during fast development cycles.

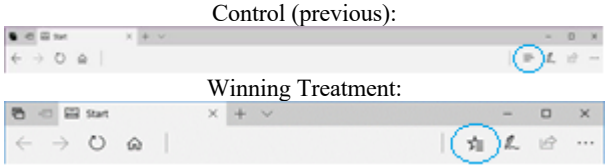

Figure 10. Previous Edge Hub icon (top) and improved designed based on experimentation (bottom)

D. The Interconnectedness of the OS, Apps, and Services

As illustrated by examples throughout this paper, the operating system is complex and interconnects with many apps and services. This leads to various challenges.

One challenge is that apps and services running on top of Windows can use different 'identifiers' to track their users and devices. For example, WExp uses the device as the identifier (i.e. the unit of randomization); for the Microsoft Office team, the unique identifier is typically a user. A single device can have multiple users and a single user can have many devices. The many-to-many mapping can introduce challenges when we try to understand the impact of Windows OS experiments on Office apps and vice versa. We typically have to assess and determine, on a case-by-case basis, the appropriate aggregations and comparisons to use.

As adoption of experimentation grows across the software industry, another challenge is that many teams (internal and external) are launching experiments that impact user experiences on Windows. This will lead to challenges in coordination, as well as in understanding all the sources of change that impact user/system behaviors. Furthermore, for individual feature teams, it can be challenging to tease out the contribution of their features, especially when potentially interacting features use different identifiers (as discussed above). As experimentation becomes more prevalent, these scenarios are likely to become even more complicated.

VIII. Conclusion

Although experimentation is quickly becoming standard best-practices for apps and online services, controlled experiments in operating systems is only starting to gain momentum. Our experiences with experimenting using WExp show that experimentation can be valuable for innovating in the OS in a safe, controlled, and data-driven manner. Thou our insights have external validity limitations as they are based on one operating system (albeit a wide-used OS of practical significance), we are not aware of other reports on experimentation in the operating system. We encourage other organizations that produce operating systems (e.g. Apple, Google) to share and compare their approaches.

We hope that our experiences can inspire practitioners to start experimenting in the OS, platform, or other non-user-facing components. When implementing their experimentation systems, we hope our insights from implementing WExp can help them design a successful system. Finally, we hope that the knowledge we contribute will help practitioners to execute better experiments and help researchers to bring forth better solutions to issues. Together, we can actualize more and better innovations for our increasingly software dependent world.

Acknowledgment

We thank all the engineers that contributed to the WExp system and all the teams that leveraged it to engineer their features. Without your work this would not be possible.

References

[1] D. Tang, A. Agarwal, D. O. Brien, and M. Meyer, "Overlapping Experiment Infrastructure : More , Better , Faster Experimentation," in *Proc SIGKDD '10*, 2010, pp. 17–26.

[2] R. Kohavi, B. Frasca, T. Crook, R. Henne, and R. Longbotham,

"Online experimentation at Microsoft," in *Prcoeedings of the Workshop on Data Mining Case Studies and Practice*, 2009.

[3] R. Kohavi and M. Round, "Front Line Internet Analytics at Amazon.com," *eMetrics Summit*, 2004. [Online]. Available: http://ai.stanford.edu/~ronnyk/emetricsAmazon.pdf .

[4] R. Kohavi, A. Deng, B. Frasca, R. Longbotham, T. Walker, and Y. Xu, "Trustworthy Online Controlled Experiments : Five Puzzling Outcomes Explained," in *Proc SIGKDD '12*, 2012, pp. 786–794.

[5] T. Whitney, M. Satran, M. Jacobs, and D. Das, "What's a Universal Windows Platform (UWP) App?," *Windows Docs*, 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide.

[6] R-core, "Student's T-Test," *RDocumentation*. [Online]. Available: https://www.rdocumentation.org/packages/stats/versions/3.5.1/topics/t.test.

[7] D. Spinellis, "A Tale of Four Kernels," in *Proc ICSE '08*, 2008, pp. 381–390.

[8] W. Vogels, "File System Usage in Windows NT 4.0," in *Proc SOSP '99*, 1999, pp. 93–109.

[9] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt, "Debugging in the (very) large: ten years of implementation and experience," in *Proc SOSP '09*, 2009, pp. 103–116.

[10] L. Dorrendorf, Z. Gutterman, and B. Pinkas, "Cryptanalysis of the Random Number Generator of the Windows Operating System," *TISSEC*, vol. 13, no. 1, 2009.

[11] S. Narayan, S. S. Kolahi, Y. Sunarto, D. D. T. Nguyen, and P. Mani, "Performance Comparison of IPv4 and IPv6 on Various Windows Operating Systems," in *Proc ICCIT '08*, 2008, pp. 663–668.

[12] E. Cota-Robles and J. P. Held, "A Comparison of Windows Driver Model Latency Performance on Windows NT and Windows 98," in *Proc SOSP '99*, 1999, pp. 159–172.

[13] S. Zhang, L. Wang, R. Zhang, and Q. Guo, "Exploratory Study on Memory Analysis of Windows 7 Operating System," in *Proc ICACTE '10*, 2010.

[14] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy, "Does Distributed Development Affect Software Quality? An Empirical Case Study of Windows Vista," in *Proc ICSE '09*, 2009, pp. 518–528.

[15] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective Data-Race Detection for the Kernel," in *Proc OSDI '10*, 2010, pp. 151–162.

[16] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance Debugging in the Large via Mining Millions of Stack Traces," in *Proc ICSE '12*, 2012, pp. 145–155.

[17] P. L. Li, R. Kivett, Z. Zhan, S. Jeon, N. Nagappan, B. Murphy, and A. J. Ko, "Characterizing the differences between pre- and post-release versions of software," *Proc ICSE '11*, pp. 716–725, 2011.

[18] A. Deng, Y. Xu, R. Kohavi, and T. Walker, "Improving the sensitivity of online controlled experiments by utilizing pre-experiment data," in *Proc WSDM '13*, 2013, p. 123.

[19] B. Ding, H. Nori, P. Li, and J. Allen, "Comparing Population Means under Local Differential Privacy: with Significance and Power," in *Proc AAAI '18*, 2018.

[20] P. Dmitriev and X. Wu, "Measuring Metrics," *Proc CIKM '16*, pp. 429–437, 2016.

[21] P. Dmitriev, S. Gupta, K. Dong Woo, and G. Vaz, "A Dirty Dozen: Twelve Common Metric Interpretation Pitfalls in Online Controlled Experiments," *Proc KDD '17*, pp. 1427–1436, 2017.

[22] H. Hohnhold, D. O'Brien, and D. Tang, "Focusing on the Long-term," in *Proc KDD '15*, 2015, pp. 1849–1858.

[23] P. Dmitriev, B. Frasca, S. Gupta, R. Kohavi, and G. Vaz, "Pitfalls of long-term online controlled experiments," in *Proc Big Data '16*, 2016, pp. 1367–1376.

[24] R. Kohavi and S. Thomke, "The Surprising Power of Online Experiments," *Harv. Bus. Rev.*, vol. 95, no. 5, p. 74, 2017.

[25] A. Fabijan, P. Dmitriev, H. H. Olsson, and J. Bosch, "The Benefits of Controlled Experimentation at Scale," in *Proc SEAA'17*, 2017, pp. 18–26.

[26] Y. Xu, N. Chen, A. Fernandez, O. Sinno, and A. Bhasin, "From Infrastructure to Culture: A/B Testing Challenges in Large Scale Social Networks," in *Proc KDD'15*, 2015, pp. 2227–2236.

[27] S. Gupta, L. Ulanova, S. Bhardwaj, P. Dmitriev, P. Raff, and A. Fabijan, "The Anatomy of a Large-Scale Online Experimentation Platform," in *Proc ICSA '18*, 2018.

[28] R. Kohavi, A. Deng, R. Longbotham, and Y. Xu, "Seven Rules of Thumb for Web Site Experimenters," in *Proc KDD '14*, 2014, pp. 1857–1866.

[29] R. Kohavi and R. Longbotham, "Online Experiments: Lessons Learned," *Computer (Long. Beach. Calif).*, vol. 40, no. 9, pp. 103–105, 2007.

[30] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empir. Softw. Eng.*, vol. 14, no. 2, pp. 131–164, 2009.

[31] P. L. Li, M. Ni, S. Xue, J. P. Mullally, M. Garzia, and M. Khambatti, "Reliability assessment of mass-market software: insights from Windows Vista®," *Proc ISSRE '08*, pp. 265–270, Nov. 2008.

[32] T. Myerson, "An Update on What's Coming Next for Windows Insiders," *Windows Insider*, 2017. [Online]. Available: https://insider.windows.com/en-us/articles/update-whats-coming-next-windows-insiders/.

[33] R. Kohavi, A. Deng, B. Frasca, T. Walker, Y. Xu, and N. Pohlmann, "Online controlled experiments at large scale," *Proc KDD '13*, p. 1168, 2013.

[34] P. Bright, "Windows 7, 8.1 Moving to Windows 10's Cumulative Update Model," *arstechnia.com*, 2016. [Online]. Available: https://arstechnica.com/information-technology/2016/08/windows-7-8-1-moving-to-windows-10s-cumulative-update-model/.

[35] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan, "TCP Fast Open," in *Proc CoNEXT '11*, 2011.

[36] D. Halfin and B. Lich, "Build Deployment Rings for Windows 10 Updates," *Windows Docs*, 2017. [Online]. Available: https://docs.microsoft.com/en-us/windows/deployment/update/waas-deployment-rings-windows-10-updates.

[37] E. Bott, "Windows 10 Telemetry Secrets: Where, When, and Why Microsoft Collects Your Data," *ZDNet*, 2016. [Online]. Available: https://www.zdnet.com/article/windows-10-telemetry-secrets/.

[38] A. Deng and X. Shi, "Data-Driven Metric Development for Online Controlled Experiments," in *Proc KDD '16*, 2016, pp. 77–86.

[39] S. Sawaya, "How Windows Insider Feedback Influences Windows 10 Development," *Windows Blogs*, 2015. [Online]. Available: https://blogs.windows.com/windowsexperience/2015/06/12/how-windows-insider-feedback-influences-windows-10-development/.

[40] K. Kniskern, "Looks like Microsoft is testing new icons for Edge Hub," *OnMSFT*, 2017. [Online]. Available: https://www.onmsft.com/news/looks-like-microsoft-is-testing-new-icons-for-edge-feedback-hub.

[41] J. Gupchup, Y. Hosseinkashi, P. Dmitriev, D. Schneider, R. Cutler, A. Jefremov, and M. Ellis, "Trustworthy Experimentation Under Telemetry Loss," in *Proc CIKM'18*, 2018, pp. 387–396.